Verified Propagation Redundancy and Compositional UNSAT Checking in CakeML

Downloaded from: https://research.chalmers.se, 2026-10-14 16:09 UTC

Citation for the original published paper (version of record):

Tan, Y., Heule, M., Myreen, M. (2023). Verified Propagation Redundancy and Compositional UNSAT Checking in CakeML. International Journal on Software Tools for Technology Transfer, 25(2): 167-184. http://dx.doi.org/10.1007/s10009-022-00690-y

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-022-00690-y

GENERAL

Special Issue: TACAS 2021

Verified Propagation Redundancy and Compositional UNSAT Checking in CakeML

Yong Kiam Tan · Marijn J. H. Heule · Magnus O. Myreen

Accepted: 4 November 2022
© The Author(s) 2023

Abstract Modern SAT solvers can emit independently-checkable proof certificates to validate their results. The state-of-the-art proof system that allows for compact proof certificates is *propagation redundancy* (PR). However, the only existing method to validate proofs in this system with a formally verified tool requires a transformation to a weaker proof system, which can result in a significant blowup in the size of the proof and increased proof validation time. This article describes the first approach to formally verify PR proofs on a succinct representation. We present (i) a new *Linear PR* (LPR) proof format, (ii) an extension of the `DPR-trim` tool to efficiently convert PR proofs into LPR format, and (iii) `cake_lpr`, a verified LPR proof checker developed in CakeML. We also enhance these tools with (iv) a new *compositional* proof format designed to enable separate (parallel) proof checking. The LPR format is backwards compatible with the existing LRAT format, but extends LRAT with support for the addition of PR clauses. Moreover, `cake_lpr` is verified using CakeML's binary code extraction toolchain, which yields correctness guarantees for its machine code (binary) implementation. This further distinguishes our clausal proof checker from existing checkers because unverified extraction and compilation tools are removed from its trusted computing base. We experimentally show that: LPR provides efficiency gains over existing proof formats; `cake_lpr`'s strong correctness guarantees are obtained without significant sacrifice in its performance; and the compositional proof format enables scalable parallel proof checking for large proofs.

Keywords linear propagation redundancy · binary code extraction · compositional proof checking

1 Introduction

Given a formula of propositional logic, the task of a SAT solver is to decide whether there exists an assignment that satisfies the formula. Such a *satisfying assignment*, if found by a SAT solver, is easily verifiable by independent checkers and so one does not need to trust the inner workings of the solver. The situation with *unsatisfiable* formulas, i.e., where no satisfying assignment exists, is not as straightforward. Here, SAT solvers must produce an *unsatisfiability proof* (also called a *refutation*) for the input formula. Ideally, the proof system and corresponding proof format for such proofs should be sufficiently expressive, allowing SAT solvers to efficiently produce proofs that correspond to the SAT solving techniques they use at runtime. At the same time, the resulting proofs ought to be efficiently checkable by independent and trustworthy tools.

The de facto standard proof system for propositional unsatisfiability proofs is Resolution Asymmetric Tautology (RAT) [31]. The associated DRAT format [55] combines clause addition based on RAT steps and clause deletion. Independent checking tools can validate proofs in the DRAT format; they have been used to check the results of the SAT competitions since 2014 [55] and in industry [20]. Enriching DRAT proofs with hints is the main technique for developing efficient verified proof checkers, e.g., existing verified checkers use the enriched proof formats LRAT [10] and GRAT [39].

A recently proposed proof system, called Propagation Redundancy (PR) [28], generalizes RAT. There exist short

The first author was supported by A*STAR, Singapore, the second author was supported by the National Science Foundation (NSF) under grant CCF-2010951, and the third author was supported by the Swedish Foundation for Strategic Research, Sweden. This work was also supported by NSF award number ACI-1445606 at the Pittsburgh Supercomputing Center (PSC).

Yong Kiam Tan and Marijn J. H. Heule
Computer Science Department, Carnegie Mellon University, Pittsburgh, USA
E-mail: yongkiat@cs.cmu.edu
E-mail: marijn@cmu.edu

Magnus O. Myreen (✉)
Chalmers University of Technology, Gothenburg, Sweden
E-mail: myreen@chalmers.se

Published online: 27 February 2023

PR proofs without new variables for many problems that are hard for resolution, such as pigeonhole formulas, Tseitin problems, and mutilated chessboard problems [26]. Due to the absence of new variables it is easier to find PR proofs automatically [27], and it is considered unlikely that there exist short RAT proofs for these problems that do not introduce new variables nor reuse eliminated variables [28]. Such PR proofs can be checked directly [28], or they can first be transformed into DRAT proofs or even Extended Resolution proofs by introducing new variables [25, 34]. In theory, the blowup is small, i.e., polynomial-sized. However, in practice, the transformed proofs can be significantly more expensive to validate compared to the original PR proofs [28].

A natural question arises: why should proof checkers be trusted to correctly check proofs if we do not likewise trust SAT solvers to correctly determine satisfiability? One answer is that proof checkers are much easier to implement so their code can be carefully audited. Another answer is that the algorithms underlying proof checkers have been *formally verified* in a proof assistant [10, 20, 39]. However, to get executable code for these verified checkers, some additional unverified steps are still required. Although unlikely, each of these steps can introduce bugs in the resulting executable: (a) the algorithms are extracted by unverified code generation tools into source code for a programming language; (b) unverified parsing, file I/O, and command-line interface code is added; (c) the combined code is compiled by unverified compilers to executable machine code.

The contributions of this article are: (i) a new Linear PR (henceforth LPR) proof format that enriches PR proofs with hints and is backwards compatible with the LRAT format, (ii) an extension of the existing `DPR-trim` tool [28] to efficiently convert PR proofs into LPR format, and (iii) `cake_lpr`, an efficient verified LPR proof checker with correctness guarantees, including for steps (a)–(c) enumerated above. The `cake_lpr` tool was used to validate the unsatisfiability proofs for the 2020 SAT Competition because of its strong trust story combined with easy compilation and usage. Moreover, the stronger PR proof system could be supported in future competitions. The tool is publicly available at: https://github.com/tanyongkiam/cake_lpr

This article extends our conference version [53] with: (iv) a new *compositional* proof format consisting of a top-level summary proof whose proof steps can be separately justified by respective underlying proofs and (v) verified extensions in `cake_lpr` to support the compositional proof format. The new correctness result for `cake_lpr` allows users to exploit verified compositional proof checking by running parallel instances of the tool to check very large unsatisfiability proofs, such as those typically found in SAT-solver aided proofs of mathematical results [23, 29, 35]. In particular, we explain how compositional proofs can be conveniently generated from the *cube-and-conquer* [22] SAT solving technique that is naturally parallelizable. Together with our verified checker, this enables a fully parallel pipeline for SAT solving, proof generation, and verified proof checking. To the best of our knowledge, this is the first verification result for a proof checker that formally accounts for multiple, separate executions of the checker.

Section 3 shows how PR proofs are enriched to obtain LPR proofs and presents the corresponding LPR proof checking algorithm (Contributions (i) & (ii)). Existing LRAT proof checkers can be extended in a clean and minimal way to support LPR proofs. Section 4 introduces the compositional proof format which extends an underlying clausal proof format with support for separate proof checking (Contribution (iv)). Section 5 explains the implementation of our proof checker in CakeML, as well as the correctness guarantees and high-level verification strategy behind the proofs (Contributions (iii) & (v)). Section 6 benchmarks our proof checker against existing implementations. A summary comparison of the new proof checker against existing verified proof checkers is in Table 1.

2 Background

This section provides background on CakeML and its related tools. It also recalls the standard problem format and clausal proof systems used by SAT solvers.

2.1 HOL4 and CakeML

HOL4 is a proof assistant implementing classical higher-order logic [51] and CakeML [45] is a programming language with syntax and semantics formally defined in HOL4. Tools for developing verified CakeML software are used to fill the verification gaps in the correspondingly enumerated items in Section 1:

(a) Two tools are used to produce (or extract) verified CakeML source code in HOL4:

Table 1 A comparison of SAT proof checkers that have been verified in various proof assistants [10, 20, 39]. Green background (cells with +) indicates, in our view, desirable properties, e.g., LPR is based on a stronger proof system than LRAT and GRAT, while red backgrounds (cells with ×) indicate less desirable properties. Yellow backgrounds (cells with −) are also undesirable but to a lesser extent.

Property	ACL2 checker [20]	Coq checker [10]	`GRATchk` [39]	`cake_lpr`
Proof System (Section 3)	− LRAT	− LRAT	− GRAT	+ LPR
Executable Code (Section 5)	− Directly Executed	× Unverified Extraction	× Unverified Extraction	+ Binary Code Extraction
Checking Speed (Section 6)	+ Fast	× Slow	+ Very Fast	+ Fast (Parallelizable)

- the CakeML proof-producing translator [46] automatically synthesizes verified source code from pure algorithmic specifications;
- the CakeML characteristic formula (CF) framework [19] provides a separation logic which can be used to manually verify (more efficient) imperative code for performance-critical parts of the proof checker.

(b) CakeML provides a foreign function interface (FFI) and a corresponding formal FFI model [15]. These are used to verify system call interactions, e.g., file I/O and command-line interfaces, under carefully specified assumptions on the system environment.

(c) Most importantly, CakeML has a compiler that is *verified* [54] to preserve the semantics of source CakeML programs down to their compiled machine-code implementations. Hence, all guarantees obtained from the preceding steps can be carried down to the level of machine code with proofs.

The combination of these tools enables *binary code extraction* [36] where verified machine code is extracted directly in HOL4. Several CakeML programs have been verified using these tools, including: certificate checkers for floating-point error bounds [6] and vote counting [18], an OpenTheory article checker [1], and the bootstrapped CakeML compiler [54]. Other toolchains can be used to build verified checkers, e.g., Œuf provides a similar binary code extraction toolchain in the Coq proof assistant [44]; the Verified Software Toolchain [9] provides a program logic for a subset of C which can be compiled with the verified compiler CompCert [40]; and the Isabelle Refinement framework can produce verified LLVM implementations [37, 38].

2.2 SAT Problems and Clausal Proofs

Fix a set of boolean *variables* $x_1, \ldots, x_n$, where the negation of variable x_i is denoted $\overline{x}_i$, and the negation of $\overline{x}_i$ is identified with x_i. Variables and their negations are called *literals* and are denoted using l.

The input for propositional SAT solvers is a formula F in *conjunctive normal form* (CNF) over the set of variables $x_1, \ldots, x_n$. Here, CNF means that F consists of an outer logical conjunction $F \equiv \bigwedge_{i=1}^{m} C_i$, where each *clause* C_i is a disjunction over some of the literals $C_i \equiv l_{i1} \vee l_{i2}, \cdots \vee l_{ik}$; in general, each clause can contain a different number of literals. Formulas in CNF can be represented directly as sets of clauses and clauses as sets of literals. The empty clause is denoted $\bot$.

An *assignment* α assigns boolean values (true or false) to each variable; α can be *partial*, i.e., it only assigns values to some of the variables. Like formulas and clauses, a (partial) assignment α can be represented as a set of literals such that $l \in \alpha$ iff α assigns l to true. For consistency, α may contain at most one of each literal l or its negation. The negation of an assignment, denoted $\overline{\alpha}$, assigns the negation of all literals in α to true. An assignment α *satisfies* a clause C iff their set intersection is nonempty. Additionally, we define $C\,|_\alpha = \top$ if α satisfies C; otherwise, $C\,|_\alpha$ denotes the result of removing from C all the literals falsified by α, i.e., $C\,|_\alpha = C \setminus \overline{\alpha}$. For a formula F, we define $F\,|_\alpha = \{C\,|_\alpha \mid C \in F \text{ and } C\,|_\alpha \neq \top\}$. Intuitively, $F\,|_\alpha$ contains the remaining simplified clauses in formula F after committing to the partial assignment α.

The task of a SAT solver is to determine whether F is *satisfiable*, i.e., whether there exists a (possibly partial) assignment α such that $F\,|_\alpha$ is empty. Any satisfying assignment can be used as certificate of satisfiability. Formulas without a satisfying assignment are *unsatisfiable*. Certifying unsatisfiability is more difficult and typically uses a *clausal* proof system [28]. The idea behind these proof systems is briefly recalled next, using the key concept of *clause redundancy*.

Definition 1 A clause C is *redundant* with respect to formula F iff $F \wedge C$ and F are both satisfiable or both unsatisfiable, i.e., they are satisfiability equivalent.

A clause C that is redundant for F can be added to F without changing its satisfiability. Clausal proof systems work by successively adding redundant clauses to F until the empty clause $\bot$ is added. Such a sequence of additions is illustrated below:

$$F \overbrace{\Longrightarrow}^{+\text{ redundant } C_1} F \wedge C_1 \overbrace{\Longrightarrow}^{+\text{ redundant } C_2} F \wedge C_1 \wedge C_2$$
$$\vdots$$
$$\underbrace{\Longrightarrow}_{+\text{ redundant } \bot} F \wedge C_1 \wedge C_2 \wedge \cdots \wedge \bot$$

Satisfiability is preserved along each $\Longrightarrow$ step because of clause redundancy, e.g., satisfiability of F implies satisfiability of $F \wedge C_1$. Since the final formula containing $\bot$ is unsatisfiable, the sequence of redundant clause addition steps $C_1, C_2, \ldots, \bot$ corresponds to a proof of unsatisfiability for F. Deciding clause redundancy is as hard as solving the SAT problem itself because $\bot$ is always redundant for unsatisfiable formulas. The difference between clausal proof systems is how the redundancy of a (proposed) redundant clause C is efficiently certified at each proof step.

Many syntactic notions of redundancy are based on unit propagation. A *unit clause* is a clause with only one literal. The result of applying the *unit clause rule* to a formula F is the formula $F|l$ where (l) is a unit clause in F. The iterated application of the unit clause rule to a formula F until no unit clauses are left is called *unit propagation*. If unit propagation on F yields the empty clause $\bot$, denoted by $F \vdash_1 \bot$, we say that F implies $\bot$ by unit propagation. The notion of *implied by unit propagation* is also used for regular clauses as follows: $F \vdash_1 C$ iff $F \wedge \neg C \vdash_1 \bot$ with $\neg C = \bigwedge_{l \in C}(\bar{l})$. Observe that $\neg C$ can be viewed as a partial assignment that assigns the literals $\bar{l}$, for $l \in C$, to true. For a formula G, $F \vdash_1 G$ iff $F \vdash_1 C$ for all $C \in G$. The main clausal proof system used in this article is based on *propagation redundant* clauses, which are defined as follows.

Definition 2 Let F be a formula, C a nonempty clause, and α the smallest assignment that falsifies C. Then, C is *propagation redundant* (PR) with respect to F if there exists an assignment ω which satisfies C and such that $F|\alpha \vdash_1 F|\omega$.

Intuitively, a PR clause C is redundant because any satisfying assignment for F that does not already satisfy C can be modified to a satisfying assignment for $F \wedge C$ by updating its literals assigned to true according to the (partial) witnessing assignment ω (see [28, Theorem 1] for more details). Propagation redundancy is efficiently checkable in polynomial time using the witnessing assignment ω and PR generalizes various other notions of clause redundancy, including the de facto standard Resolution Asymmetric Tautology (RAT) proof system (see [28, Theorem 2]) that is able to compactly express all current techniques used in state-of-the-art SAT solvers [31]. There is ongoing research towards integrating PR in SAT solvers [27, 28]. For example, PR-based preprocessing has been shown to improve solver performance on SAT competition benchmarks [49].

In practice, clausal proof formats also support clause deletions to speed up proof validation, especially for proof checking steps that need to iterate over the entire formula, e.g., $F|\alpha \vdash_1 F|\omega$ for Def. 2. Hence, unsatisfiability proofs for formula F are modeled as sequences $I_1, \ldots, I_n$ of *instructions* that either add or delete a clause; an *addition instruction* is a triple $\langle \mathsf{a}, C, \omega \rangle$, where C is a clause and ω is a (possibly empty) *witnessing assignment*; a *deletion instruction* is a pair $\langle \mathsf{d}, C \rangle$ where C is a clause. The sequence $I_1, \ldots, I_n$ gives rise to formulas $F_1, \ldots, F_n$ with $F_0 = F$, where F_j is the *accumulated formula*, i.e., a (multi)set of clauses, up to the j-th instruction computed recursively according to (1).

$$F_j = \begin{cases} F_{j-1} \cup \{C_j\} & \text{if } I_j \text{ is of the form } \langle \mathsf{a}, C_j, \omega \rangle \\ F_{j-1} \setminus \{C_j\} & \text{if } I_j \text{ is of the form } \langle \mathsf{d}, C_j \rangle \end{cases} \quad (1)$$

A PR proof of unsatisfiability is *valid* if the last instruction adds the empty clause $I_n = \langle \mathsf{a}, \bot, \emptyset \rangle$, and, for all addition instructions $I_j = \langle \mathsf{a}, C_j, \omega_j \rangle$, it holds that C_j is PR with respect to F_{j-1} using witness ω_j. In case an empty witness is provided for I_j, then $F_{j-1} \vdash_1 C_j$ should hold.

3 Linear Propagation Redundancy

This section describes a new clausal proof format called Linear Propagation Redundancy (LPR) which is designed to enable efficient validation of PR clauses using a (verified) proof checker. It also presents our enhancements to the `DPR-trim` tool[1] to efficiently add hints to PR proofs, thereby turning them into LPR proofs. Throughout the section, we emphasize how LPR can be viewed as a clean and minimal extension of the existing LRAT proof format, which thereby enables its straightforward implementation in existing LRAT tools.

The most commonly used proof format for SAT solvers is DRAT, which combines deletion with RAT redundancy [55]. DRAT proofs are easy for SAT solvers to emit and top-tier SAT solvers support it, but they have some disadvantages

[1] LPR hint addition is now part of the public GitHub version of `DPR-trim` using the command-line option `-L` at: https://github.com/marijnheule/dpr-trim.

$$
\begin{array}{lcl}
\langle proof \rangle & = & \{\langle line \rangle\} \\
\langle line \rangle & = & (\langle lpr \rangle \mid \langle delete \rangle), \text{“}\backslash n\text{”} \\
\langle lpr \rangle & = & \langle id \rangle, \langle clause \rangle, \langle \boldsymbol{witness} \rangle, \text{“0”}, \langle idlist \rangle, \\
 & & \{\langle reduced \rangle\}, \text{“0”} \\
\langle delete \rangle & = & \langle id \rangle, \text{“}d\text{”}, \langle idlist \rangle, \text{“0”} \\
\langle reduced \rangle & = & \langle neg \rangle, \langle idlist \rangle \\
\langle idlist \rangle & = & \{\langle id \rangle\} \\
\langle id \rangle & = & \langle pos \rangle \\
\langle lit \rangle & = & \langle pos \rangle \mid \langle neg \rangle \\
\langle pos \rangle & = & \text{“1”} \mid \text{“2”} \mid \ldots \\
\langle neg \rangle & = & \text{“}-\text{”}, \langle pos \rangle \\
\langle clause \rangle & = & \{\langle lit \rangle\} \\
\boldsymbol{\langle witness \rangle} & \boldsymbol{=} & \boldsymbol{\{\langle lit \rangle\}}
\end{array}
$$

Fig. 1 The grammar for the LPR format. Additions compared to the LRAT grammar [10] are highlighted in **bold**.

for verified proof checking. In particular, checking whether a clause is RAT requires a significant amount of proof search to find the unit clauses necessary for showing the implied-by-unit-propagation $\vdash_1$ property. This complicates verification of the proof checking algorithm and slows down the resulting verified proof checkers. The idea behind the Linear RAT (LRAT) [10, 20] and GRAT [39] formats is to include these unit clauses as hints so that verified proof checkers can follow the hints directly without the need for proof search. The LPR format lifts this idea to allow fast validation of the PR property as follows.

An assignment ω *reduces* a clause C if $C|_\omega \subset C$ and $C|_\omega \neq \top$. To check the PR property $F|_\alpha \vdash_1 F|_\omega$, it suffices to check, for each clause $C \in F$ reduced by ω, that $F|_\alpha \vdash_1 C|_\omega$. In practice, a smaller ω yields a cheaper PR check. The LPR format extends the PR format by adding, for each clause reduced by the witness, a list of all unit clause hints required for showing the implied-by-unit-propagation property. Additionally, in order to point to clauses, the LPR format includes an index for each clause at the beginning of each line. The grammar of the LPR format is shown in Fig. 1.

The `DPR-trim` tool [28, Sect. 6] is built on top of `DRAT-trim` and facilitates verification of DPR proofs, a generalization of DRAT proofs with PR clause addition. If a clause addition step includes a witnessing assignment ω, then the PR redundancy check is performed. Otherwise `DPR-trim` falls back on the RAT check from the `DRAT-trim` code base. Deletion steps are not validated. `DPR-trim` has similar features compared to `DRAT-trim`, including backward checking, extraction of unsatisfiable cores, and proof optimization.

Our extension to `DPR-trim` enriches input PR proofs by finding and adding all required unit clause hints to produce LPR proofs. Most of the changes generalize the code to produce LRAT proofs in `DRAT-trim`. The main PR-specific optimization shrinks the witness ω where possible: every literal in $\omega \cap \alpha$ is removed as well as any literal in ω that is implied by unit propagation from $F|_\alpha$. The shrinking was shown to be correct [28], but has not been implemented so far. We observed that the witnesses in the PR proofs produced by `SaDiCaL` [27] can be substantially compressed using this method.

Fig. 2 (left) shows an example formula in the standard DIMACS problem format. The DIMACS format includes a header line starting with “`p cnf` ” followed by the number of variables and the number of clauses. The non-comment lines (not starting with “`c` ”) represent clauses, and they end with “`0`”. Positive integers denote positive literals, while negative integers denote negative literals. By convention (following LRAT), the clauses are implicitly indexed according to their order of appearance in the file, starting from index 1. Fig. 2 (right) shows a corresponding proof in LPR format.[2] Deletion lines in LPR are formatted identically to LRAT [10] (not shown here). For clause addition lines, the LPR format only differs from LRAT in case the clause to be added has PR but not RAT redundancy. A clause addition line in LPR format consists of three parts. The first part is the first integer on the line, which denotes the index of the new clause. The second part exactly matches the PR proof format [28]. It consists of the redundant clause and its witness; the first group of literals is the clause while the (potentially empty) witness starts from the second occurrence of the first literal of the clause until the first `0` that separates the unit clause hints. The third part (after the first `0`) are the unit clause hints, which exactly matches the LRAT format [10].

The checking algorithm for LPR, shown in Fig. 3, overlaps significantly with that for LRAT (see [10, Algorithm 1]). The only differences are in Steps 4 and 5.1. In Step 4, the witness is used (if present) instead of always using the first literal in C_j. In Step 5.1, clauses are skipped if they are satisfied by the witness. Notice that a clause can only be both reduced and satisfied by a witness if the witness consists of at least two literals, while in the LRAT format witnesses always consist of exactly one literal. Note also that the algorithm does not check whether $C_j|_\omega = \top$, which is a requirement for PR. This omission is allowed because the first literal in ω in the LPR (and PR) format is syntactically the same as the first literal in C_j.

[2] The 12 line PR proof in Fig. 2 (right) can be converted to a longer 264 line RAT proof using earlier techniques [27] (not shown for brevity); to the best of our knowledge, there are no known short RAT proofs for these problems that do not introduce new variables nor reuse eliminated variables [28].

DIMACS file

```
p cnf 12 22
  1   2    3 0
  4   5    6 0
  7   8    9 0
 10  11   12 0
 -1  -4 0
 -2  -5 0
 -3  -6 0
 ...
 -7 -10 0
 -8 -11 0
 -9 -12 0
```

LPR proof file

```
23 -3 -10 -3 -10 1 12 0 -5 17  -8 20 -19  7 -22 10 0
24 -3 -11 -3 -11 2 12 0 -6 18  -9 21 -19  7 -22 10 0
25    -3                0                23 24   4 13 0
26 -6 -10 -6 -10 4 12 0 -5 11 -13  7 -14 20 -22 16 0
27 -6 -11 -6 -11 5 12 0 -6 12 -13  7 -15 21 -22 16 0
28    -6                0                26 27   4 19 0
29 -9 -10 -9 -10 7 12 0 -8 11 -13 10 -14 17 -19 16 0
30 -9 -11 -9 -11 8 12 0 -9 12 -13 10 -15 18 -19 16 0
31    -9                0                29 30   4 22 0
32    -2                0    6   9 28  31  2   3 14 0
33    -5                0    6  15 25  31  1   3  8 0
34                      0   25  28 32  33  1   2  5 0
```

Fig. 2 (Left) Clauses of an unsatisfiable pigeonhole formula (4 pigeons, 3 holes) in the DIMACS format used by SAT solvers. The first 4 clauses encode that each pigeon belongs to at least one hole, e.g., the variable 1 (resp. 4, 7, 10) is set to true iff pigeon A (resp. B, C, D) is in the first hole; the latter 18 clauses encode that no two pigeons share the same hole, e.g., the clause `-1 -4` encodes that pigeons A and B do not share the first hole. (Right) The LPR refutation consisting of clause-witness pairs and unit clause hints. The first **bold** integer in each line is the clause index while other **bold** integers are the unit clause hints. Dropping the **bold** integers yields a proof in the PR format. Redundant spaces have been added to improve readability.

Input: CNF $F = \{C_i\}_{i\in\mathcal{I}}$ and line ℓ an LPR step.
Output: `YES` if parsed clause C_j proved PR for F by ℓ, `NO` otherwise.

1. parse ℓ as $\left[j, C_j, \boldsymbol{\omega_j}, 0, \widetilde{i^0}, \{-i^k, \widetilde{i^k}\}_{k=1}^{n}\right]$, instantiating variables with (vectors of) positive integers.
2. set $\alpha \leftarrow \neg C_j$
3. for $i \in \widetilde{i^0}$
   3.1. set $C_i' \leftarrow C_i \,|\, \alpha$
   3.2. if $C_i' = \bot$, return `YES`
   3.3. if $C_i' = \top$ or $|C_i'| \geq 2$, return `NO`
   3.4. set $\alpha \leftarrow \alpha \cup C_i'$
4. **if** $\boldsymbol{\omega_j} \neq \emptyset$ **then set** $\boldsymbol{\omega} \leftarrow \boldsymbol{\omega_j}$ **else** set $\omega \leftarrow (C_j)_1$ (if $C_j = \bot$, return `NO`)
5. for $i \in \mathcal{I}$
   5.1. if C_i is **satisfied by** $\boldsymbol{\omega}$ or is not reduced by ω, skip to next iteration of Step 5.
   5.2. find k such that $i^k = i$ (from ℓ) (return `NO` if no such k exists)
   5.3. if $C_i \,|\, (\alpha \setminus \overline{\omega}) = \top$, skip
   5.4. set $\alpha' \leftarrow \alpha \cup (\neg C_i \setminus \omega)$
   5.5. for $m \in \widetilde{i^k}$
      5.5.1. set $C_m' \leftarrow C_m \,|\, \alpha'$
      5.5.2. if $C_m' = \bot$, skip to next iteration of Step 5.
      5.5.3. if $C_m' = \top$ or $|C_m'| \geq 2$, return `NO`
      5.5.4. set $\alpha' \leftarrow \alpha' \cup C_m'$
   5.6. return `NO`
6. return `YES`

Fig. 3 Algorithm to check a single clause addition step in the LPR format. The **bold** parts show the additions compared to LRAT proof checking [10].

4 Compositional Proofs

This section presents a new *compositional* proof format for unsatisfiability proof checking, motivated by the need to check very large unsatisfiability proofs behind SAT-solver aided proofs of mathematical results [23, 29, 35]. The rules of compositional proofs in propositional logic have been discussed in earlier work [24]; here, we define a format for compositional proofs and present a proof checking framework for the entire toolchain. Intuitively, the format consists of a top-level summary proof and a set of underlying proofs which are used to justify the top-level proof steps, see Fig. 4 for an illustration (explained below). The underlying proofs can be checked separately and *in parallel* to speed up proof checking. Section 4.1 presents the proof format and its proof checking algorithm. Section 4.2 explains our technique for generating compositional proofs from the *cube-and-conquer* SAT solving technique [22].

4.1 Compositional Proof Checking

Compositional proofs are modeled using instruction sequences $I_1, \ldots, I_n$ similar to clausal proofs (Section 2.2) except addition instructions are of the form $\langle \mathsf{a}, C \rangle$, i.e., they do not carry witnesses. The accumulated formulas F_j are defined according to equation 1.

The key idea behind compositional proof checking is to justify a *range* of instructions simultaneously using an underlying clausal proof, see Fig. 4. More precisely, given a pair of indices (i, j), with $i \leq j$, the algorithm checks that satisfiability of the accumulated formula F_i implies satisfiability of the accumulated formula F_j. By transitivity of satisfiability implication, this means that a proof with n instructions can be checked by separately checking k ranges (i_0, i_1), (i_1, i_2), $\ldots$, (i_{k-1}, i_k) for some choice of indices $0 = i_0 \leq i_1 \leq i_2 \leq \cdots \leq i_k = n$. If all k checks succeed, then satisfiability of F implies satisfiability of F_n and, in particular, if F_n also contains $\bot$ then the input formula F is unsatisfiable. Satisfiability implication for each pair F_i, F_j is checked using an underlying clausal proof and its

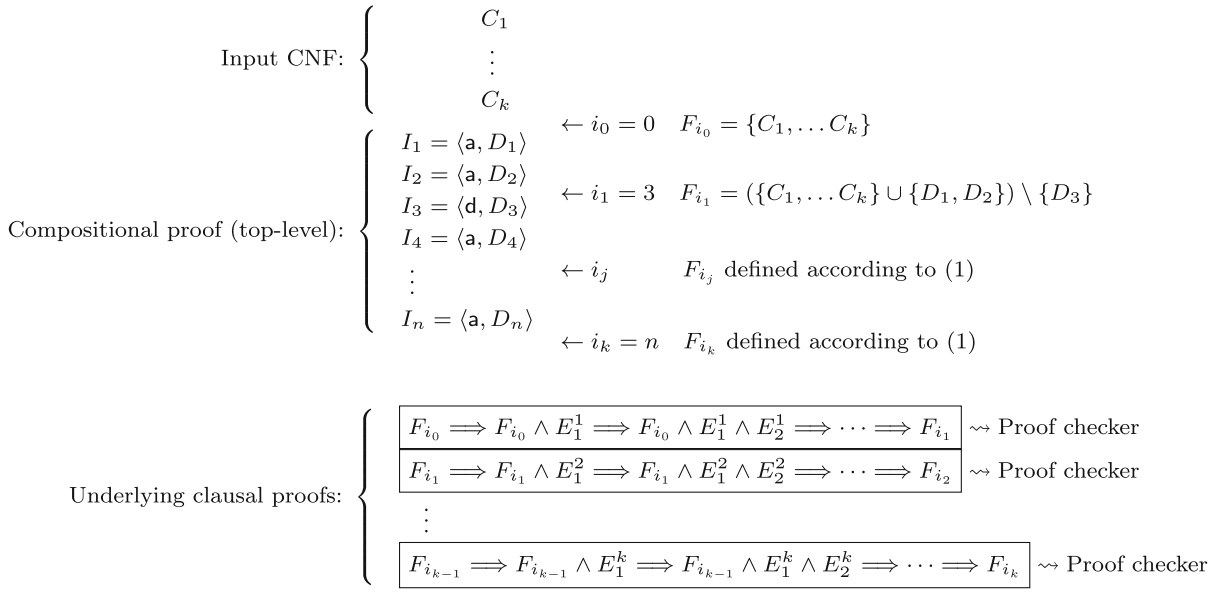

Fig. 4 Illustration of a compositional proof: the top-level proof is a sequence of instructions $I_1, \ldots, I_n$ that can add or delete clauses; the indexes $i_0, \ldots, i_k$ divide the top-level proof steps into ranges; and the boxes indicate underlying clausal proofs which are separately checked (indicated by ⇝) using instances of appropriate proof checkers, e.g., `cake_lpr`.

Input: CNF $F = \{C_i\}_{i \in \mathcal{I}}$, compositional proof P,
range (i, j), and underlying clausal proof p.
Output: `YES` if satisfiability of formula F_i is proved
to imply satisfiability of F_j by p,
`NO` otherwise.
1. set $F_0 \leftarrow F$ and parse P as an instruction sequence $I_1, \ldots, I_n$ in DRAT format.
2. if $i > j$ or $j > n$, return `NO`
3. compute accumulated formulas F_i, F_j by (1).
4. run underlying proof checker on p and F_i.
   4.1. if underlying proof checking fails, return `NO`
   4.2. set G to the resulting accumulated formula
5. if $F_j \not\subseteq G$, return `NO`
6. return `YES`

Fig. 5 Compositional proof checking algorithm for range (i, j) based on an underlying clausal proof format.

corresponding proof checker. A formal description of the algorithm for checking range (i, j) is in Fig. 5. By design, there is significant flexibility in how the underlying proofs can be checked:

- The indexes $i_0, i_1, \ldots, i_k$ can be chosen arbitrarily by tools, as long as they cover all steps of the top-level proof with $i_0 = 0$, $i_k = n$.
- Different clausal proof formats and corresponding proof checkers can be used for each underlying proof.
- The underlying proofs can be checked separately, in any order, and in parallel by running multiple instances of proof checkers on several machines.

$$
\begin{array}{lcl}
\langle proof \rangle & = & \{\langle line \rangle\} \\
\langle line \rangle & = & (\langle comp \rangle \mid \langle delete \rangle), \text{“}\backslash n\text{”} \\
\langle comp \rangle & = & \langle clause \rangle, \text{“0”} \\
\langle delete \rangle & = & \text{“}d\text{”}, \langle comp \rangle, \text{“0”} \\
\langle lit \rangle & = & \langle pos \rangle \mid \langle neg \rangle \\
\langle pos \rangle & = & \text{“1”} \mid \text{“2”} \mid \ldots \\
\langle neg \rangle & = & \text{“}-\text{”}, \langle pos \rangle \\
\langle clause \rangle & = & \{\langle lit \rangle\}
\end{array}
$$

Fig. 6 The grammar for the DRAT format [55], re-used for compositional proofs. Each $\langle line \rangle$ represents either addition (no prefix) or deletion (prefixed by “d”) of a clause.

To enable re-use of existing parsing tools, compositional proofs are syntactically represented using the DRAT format [55], recalled in Fig. 6. Clauses are implicitly numbered according to their order of addition, continuing from the last (implicit) index of the corresponding DIMACS CNF. For example, starting from a DIMACS file with 5 clauses (indices 1–5), the first added clause in the compositional proof has index 6, the second has index 7, and so on. Deletion steps are ignored for the purposes of indexing.

4.2 Compositional Proof Generation

Compositional proofs can be generated particularly conveniently from *cube-and-conquer* SAT proofs [22]. Given an input formula F, the basic idea behind cube-and-conquer is to partition F into a set of subformulas $F \wedge G_1, F \wedge G_2, \ldots, F \wedge G_n$, and to solve each subproblem separately. Here, each $G_i = l_1 \wedge l_2 \wedge \cdots \wedge l_j$ is a *cube*, i.e., a conjunction of literals, such that the disjunction of cubes $\bigvee_{i=1}^{n} G_i$ is a tautology.

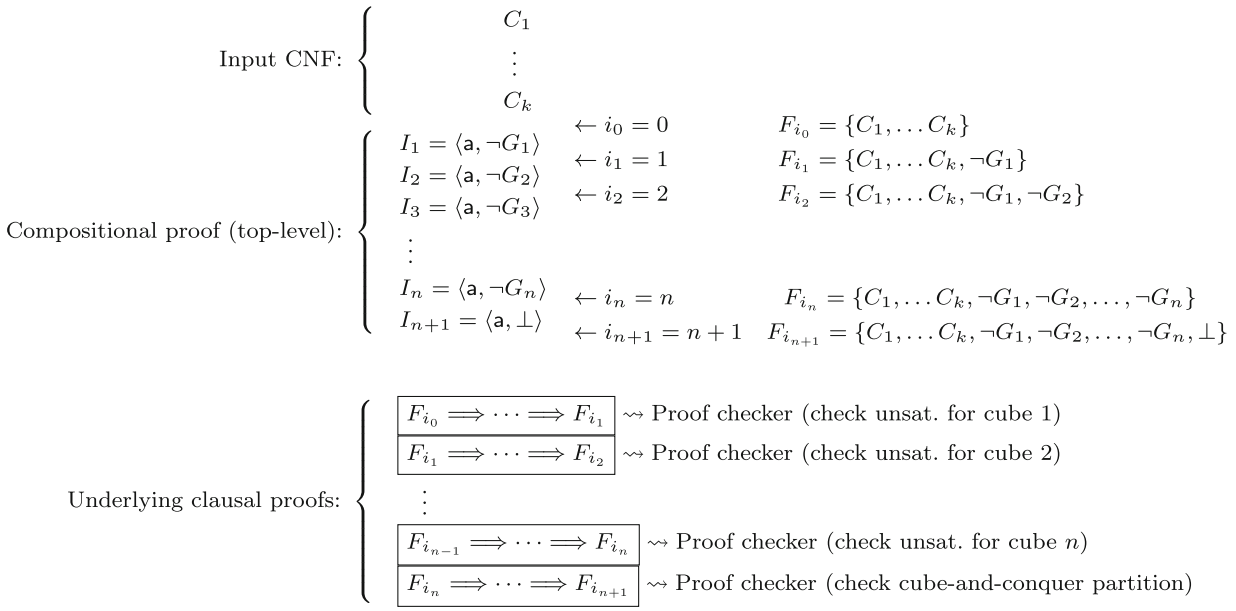

Fig. 7 Compositional proof used for the cube-and-conquer parallel SAT solving strategy. The boxes correspond to underlying proofs generated by `DRAT-trim` for each cube which can be checked using `cake_lpr` or other LRAT tools [10]. In practice, the cubes G_i used in the top-level proof steps can be simplified to core subsets $H_i \subseteq G_i$ obtained from incremental SAT solving.

Thus, if every subproblem $F \wedge G_i$ is unsatisfiable, then F is unsatisfiable.

The compositional unsatisfiability proof for F under cube-and-conquer is shown in Fig. 7, analogous to Fig. 4. Here, the top-level proof is the sequence of n additions of negated cubes $\neg G_i$ (which are clauses), followed by addition of the empty clause. The first n underlying proofs justify unsatisfiability of F under each cube, while the last proof shows that the cube-and-conquer strategy correctly partitions the space of assignments.

The top-level proof steps can be simplified while generating the underlying clausal proofs for each step. In practice, each subproblem $F \wedge G_i$ is tackled using incremental SAT solvers that support solving F under assumptions (or partial assignments) G_i. Such assumptions are typically provided either via iCNF files [56] or via the solver API; an example formula with three cubes expressed in iCNF (DIMACS files extended with assumptions) is shown in Fig. 8. If the formula F is found to be unsatisfiable under assumptions G_i, incremental solvers can further compute a subset of the cube $H_i \subseteq G_i$ that was involved in determining unsatisfiability [13]. The negation $\neg H_i$ can be used in the compositional proof in place of $\neg G_i$, e.g., the topmost cube in Fig. 8 has a proof with $H_i \subset G_i$. Since existing solvers with incremental support can only log proofs in the DRAT format, we modified `DRAT-trim` to convert such partial proofs into LRAT proofs that end with the addition of clause $\neg H_i$. In addition, when generating the LRAT proof for $\neg H_i$, the compositional proof checking algorithm Fig. 5 allows us to assume all previously added clauses, i.e., starting with

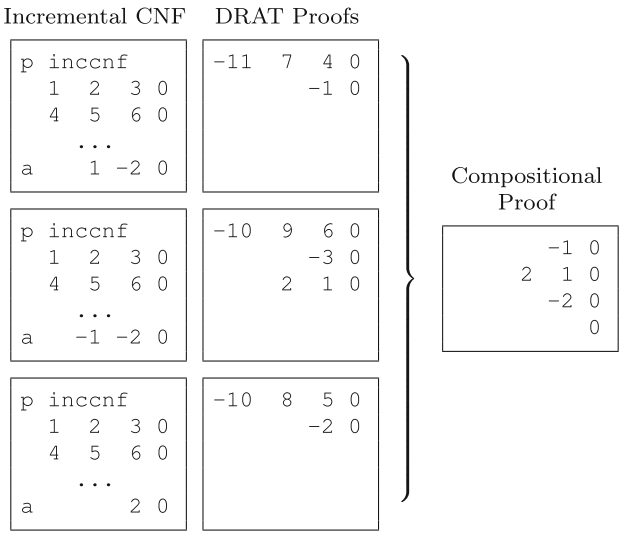

Fig. 8 (Left) Incremental CNF files for the pigeonhole formula from Fig. 2 split using three cubes (assumptions are specified using `a ... 0` lines). (Middle) Generated DRAT proofs for each incremental CNF; the last line in each proof is $\neg H_i$. (Right) The top-level compositional proof for the pigeonhole formula in Fig. 2 built from $\neg H_i$.

$F \wedge \neg H_1 \wedge \cdots \wedge \neg H_{i-1}$, which may help to simplify the LRAT proof.

The final line in compositional unsatisfiability proofs is the addition of the empty clause. In most use cases, there exists a short justification of the empty clause using the added clauses in the compositional proof, e.g., the conjunction of preceding clauses in the compositional proof is typically

unsatisfiable (in our case, by construction). Thus, a SAT solver can be used on that formula to produce the justification of the empty clause in the appropriate clausal proof format. Alternatively, if the compositional proof was constructed using more complicated cube-and-conquer strategies [22], then the underlying tree structure used to generate the cubes can be used to guide the resolution steps for obtaining the empty clause.

5 CakeML Proof Checking

This section explains the implementation and verification of `cake_lpr`, our verified CakeML LPR proof checker. Section 5.1 focuses on the high-level verification strategy which we used to reduce the verification task to mostly routine low-level proofs (details of the latter are omitted). Section 5.2 explains the main verified theorems for the proof checker. Section 5.3 highlights some of the verified performance optimizations used in the proof checker.

5.1 Implementation and Verification Strategy

The development of `cake_lpr` proceeds in three refinement steps, where each step progressively produces a more concrete and performant implementation of the proof checker. These refinements are visualized in the three columns of Fig. 9 for LPR proof checking. A similar verification process was used to add support for compositional proof checking.

Step 1 formalizes the definition of CNF formulas and their unsatisfiability, as well as the PR proof system described in Section 2.2. The inputs and outputs to the proof system are abstract and not tied to any concrete representation at this step. For example, input variables are drawn from an arbitrary type α, while clauses and CNFs are represented using sets. The correctness of the PR proof system is proved in this step, i.e., we show that a valid PR proof implies unsatisfiability of the input CNF. The proof essentially follows [28, Theorem 1].

Step 2 implements a purely functional version of the LPR proof checking algorithm from Fig. 3. Here, the inputs and outputs are given concrete representations with computable datatypes, e.g., literals are integers (similar to DIMACS), clauses are lists of integers, and CNFs are lists of clauses. These concrete representations lift naturally to the abstract, set-based representation from Step 1. The output is a `YES` or `NO` answer according to the algorithm from Fig. 3. The correctness theorem for Step 2 shows that LPR proof checking correctly refines the PR proof system, i.e., if it outputs `YES`, then there exists a valid PR proof for the input (lifted) CNF; by Step 1, this implies that the CNF is unsatisfiable. If the output is `NO`, the input CNF could still be unsatisfiable, but the input LPR proof is not valid according to the algorithm in Fig. 3.

Step 3 uses imperative features in the CakeML source language, e.g., (byte) arrays and exceptions, to improve code performance; these optimizations are detailed further in Section 5.3. This step also adds user interface features like parsing and file I/O so that the input CNF formula is read (and parsed) from a file, and the results are printed on the standard output and error streams. The verification of this step uses CakeML's proof-producing translator [46] and characteristic formula framework [19] to prove the correctness of the source code implementation of `cake_lpr`; this code is subsequently compiled with the verified CakeML compiler [54]. Composing the correctness theorem for source `cake_lpr` with CakeML's compiler correctness theorem yields the corresponding correctness theorem for the `cake_lpr` binary.

At the point of writing, the verified `cake_lpr` binary can be invoked from the command-line in five ways, each with associated soundness proofs (Section 5.2):

1. `cake_lpr <DIMACS>`
   Parses the input file in DIMACS format and prints the parsed formula.
2. `cake_lpr <DIMACS> <LPR>`
   Runs LPR proof checking on the parsed formula.
3. `cake_lpr <DIMACS1> <LPR> <DIMACS2>`
   Runs LPR transformation checking on the parsed formulas, i.e., `DIMACS1` is checked to imply `DIMACS2` (satisfiability-wise) using the LPR proof [10].
4. `cake_lpr <DIMACS> <COMP> i-j <LPR>`
   Runs compositional proof checking on the parsed formula and compositional proof `COMP` for the range `i–j` and underlying LPR proof `LPR`.
5. `cake_lpr <DIMACS> <COMP> -check <OUTPUT>`
   Checks the combined result of several compositional proof checker executions (see below).

Recall from Section 4.1 that `cake_lpr` needs to be executed (with option 4) for a set of ranges covering the lines of the compositional (top-level) proof `COMP` and we expect users to exploit this compositionality by running separate instances of `cake_lpr` in parallel on several machines. Accordingly, an important *correctness caveat* for compositional proof checking is that users correctly set up separate executions for their respective systems.

Option 5 is designed to add a simple layer of protection against user error when setting up separate (or parallel) executions. In particular, `cake_lpr` outputs the following string for each successful run of option 4, where `md5` takes the MD5 hash of the input files:

```
s VERIFIED RANGE md5(DIMACS) md5(COMP) i-j
```

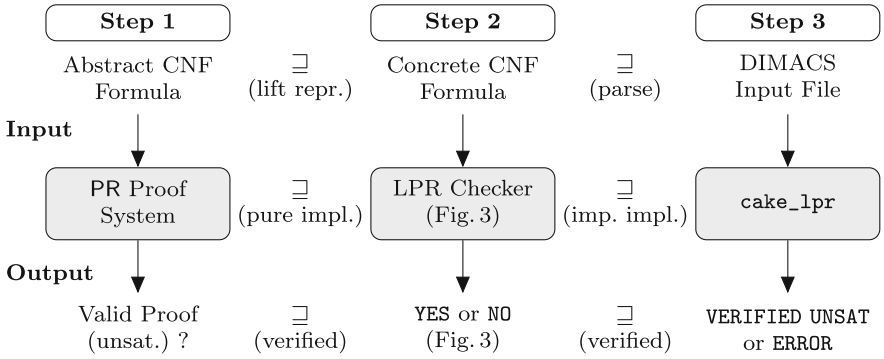

Fig. 9 The three step refinement used in the development of `cake_lpr`.

The MD5 hash is chosen for convenience because it is available on most machines, so users can, e.g., manually compare `md5` hashes of their input DIMACS and proof files to check that the correct files were used on all machines. By concatenating these outputs into an output file `OUTPUT` and executing `cake_lpr` with option 5, `cake_lpr` can be used to check that the output file contains the correct hashes and that the specified ranges appropriately cover the entire compositional proof. Our implementation checks range coverage (e.g., that ranges 0–2, 4–8, 2–4, 8–12 can be strung together to check range 0–12) by reusing a verified reachability checking function originally developed for a verified compiler optimization in the CakeML compiler [32].

5.2 Correctness Theorems

The main correctness theorem for `cake_lpr` in HOL4 is shown in Fig. 10. The first line (2) (in red) summarizes routine assumptions for compiled CakeML programs that use its basis library. Briefly, it assumes that the command-line *cl* and file system *fs* models are well-formed and the compiled code is placed in (and executed from) code memory of a machine state *ms* according to CakeML's x64 machine model *mc*.

The first guarantee (3) (in blue) is that the machine-code implementation always terminates normally according to CakeML's x64 machine-code semantics. Notably, this means the binary *never crashes* and it may emit some I/O events when run; however, it possibly terminates with an out-of-memory error (extend_with_resource_limit) if the CakeML runtime runs out of stack or heap space [54]. The second guarantee (4) (in green) states that the only observable change to the filesystem after executing `cake_lpr` are some strings printed on standard output *out* and standard error *err*. To minimize user confusion, `cake_lpr` is designed to print all error messages to standard error and only success messages on standard output.

Finally, lines (5) (in black) list the output behaviors of `cake_lpr` for each command-line option.[3]

1. When `cake_lpr` is given one command-line argument (length *cl* = 2)[4], it attempts to read and parse the file before printing (if successful) the parsed formula to standard output. The DIMACS parser (parse_dimacs) is proved to be left inverse to the DIMACS printer (print_dimacs) as follows:

   ⊢ every wf_clause *fml* ⇒
   parse_dimacs (print_dimacs *fml*) = Some *fml*

   This says that for any well-formed formula *fml*, printing that formula into DIMACS format then parsing it yields the formula back. All parsed formulas are proved to be well-formed (not shown here).
2. If two arguments are given (length *cl* = 3), then if the string "`s VERIFIED UNSAT\n`" is printed onto standard output, `cake_lpr` was provided with a file (in its first argument), and the file parses in DIMACS format to a formula *fml* whose lifted representation (interp *fml*) is unsatisfiable.

3–5. The specifications for the remaining cases have a similar flavor and are omitted here for brevity.

The correctness theorem for `cake_lpr`'s compositional proof checking is shown in Fig. 11. The definition check_successful_par characterizes a successful set of runs of `cake_lpr` using command line options 4 and 5 on input strings *fmlstr* and *pfstr*. The lines in red (6) say that there is a list of output strings *outs* such that every entry of this list is produced from the standard output of an execution of `cake_lpr` with command-line option 4 (with appropriate setup of each

[3] These lines feature HOL4 definitions for interacting with the filesystem *fs*, e.g., inFS_fname *fs s* says the string *s* is a valid filename in *fs*, all_lines *fs s* returns the file contents as a list of strings (per line), and file_content *fs s* returns the file content as a raw string.

[4] By convention, the default (zeroth) command-line argument is always the name of the executable.

```
⊢ cake_lpr_run cl fs mc ms ⇒                                                    } (2)
  machine_sem mc (basis_ffi cl fs) ms ⊆                                         }
   extend_with_resource_limit { Terminate Success (cake_lpr_io_events cl fs) } ∧ } (3)
  ∃ out err.                                                                    }
   extract_fs fs (cake_lpr_io_events cl fs) = Some (add_stdout (add_stderr fs err) out) ∧ } (4)
   if length cl = 2 then                                                        }
    if inFS_fname fs (el 1 cl) then                                             }
     case parse_dimacs (all_lines fs (el 1 cl)) of                              }
       None ⇒ out = «»                                                          }
     | Some fml ⇒ out = concat (print_dimacs fml)                               }
    else out = «»                                                               }
   else if length cl = 3 then                                                   } (5)
    if out = «s VERIFIED UNSAT\n» then                                          }
     inFS_fname fs (el 1 cl) ∧                                                  }
     ∃ fml.                                                                     }
       parse_dimacs (all_lines fs (el 1 cl)) = Some fml ∧                       }
       unsatisfiable (interp fml)                                               }
    else out = «»                                                               }
   else ...                                                                     }
```

Fig. 10 The end-to-end correctness theorem for the CakeML LPR proof checker. The cases for command-line options 1 and 2 are shown here (other cases are elided with ... for brevity).

```
check_successful_par fmlstr pfstr ≝
∃ outs.
 (∀ out. MEM out outs ⇒                                                         }
  ∃ cl fs mc ms err.                                                            }
   cake_lpr_run cl fs mc ms ∧ length cl = 5 ∧ inFS_fname fs (el 1 cl) ∧ inFS_fname fs (el 2 cl) ∧ } (6)
   file_content fs (el 1 cl) = Some fmlstr ∧ file_content fs (el 2 cl) = Some pfstr ∧ }
   extract_fs fs (cake_lpr_io_events cl fs) = Some (add_stdout (add_stderr fs err) out)) ∧ }
 ∃ cl fs mc ms.                                                                 }
  cake_lpr_run cl fs mc ms ∧ length cl = 5 ∧ inFS_fname fs (el 1 cl) ∧ inFS_fname fs (el 2 cl) ∧ }
  file_content fs (el 1 cl) = Some fmlstr ∧ file_content fs (el 2 cl) = Some pfstr ∧ all_lines fs (el 4 cl) = outs ∧ } (7)
  extract_fs fs (cake_lpr_io_events cl fs) = Some (add_stdout fs                 }
      (concat [ «s VERIFIED INTERVALS COVER 0-» ; toString (length (lines_of (strlit pfstr))); «\n» ])) }

⊢ parse_dimacs (lines_of (strlit fmlstr)) = Some fml ∧ parse_proof (lines_of (strlit pfstr)) = Some pf ∧ }
  check_successful_par fmlstr pfstr ⇒                                           } (8)
  satisfiable (interp fml) ⇒ satisfiable (interp (run_proof fml pf))            }
```

Fig. 11 The correctness theorem for compositional proof checking.

machine's filesystem and command-line arguments).[5] The lines in blue (7) say that *outs* is successfully checked by an execution of `cake_lpr` with command-line option 5 and the corresponding success string (concat [...]) is printed to standard output. Using this definition, the correctness theorem (8) says that on a successful set of runs, satisfiability of the formula *fml* parsed from *fmlstr* implies satisfiability of the final formula run_proof *fml pf* obtained by running all lines of the parsed proof *pf* on *fml*, i.e., if *fml* is F_0 and *pf* has n instructions, then run_proof *fml pf* computes the accumulated formula F_n as defined in Section 4.1.

[5] In practice, this models users concatenating the outputs of multiple `cake_lpr` executions into a single file.

The theorems in Figures 10 and 11 have a trusted computing base (TCB) where the CakeML compiler itself is not present. The TCB does, however, still include CakeML's model of the foreign function interface and assumed semantics for the targeted hardware platform. CakeML's TCB is discussed in detail in prior publications [36, 42].

5.3 Verified Optimizations

To minimize verification effort, CakeML's imperative features are only used for the most performance-critical steps of `cake_lpr`. Our design decisions are based on empirical observations about the LPR proof checking algorithm. These are explained below with reference to specific steps in the algorithm outlined in Fig. 3.

5.3.1 Array-based representations

In practice, many LPR proof steps do not require the full strength of a PR (or RAT) clause. Hence, a large part of proof checking time is spent in the Step 3 loop of the algorithm (reverse unit propagation) and it is important to compute the main loop bottleneck, $C_i|_\alpha$ in Step 3.1, as efficiently as possible. Here, CakeML's native byte arrays are used to maintain a compact bitset-like representation of the assignment α, so that $C_i|_\alpha$ can be computed in one pass over C_i with constant time bitset lookup for each literal in C_i.

For proof steps requiring the full strength of PR clauses, Step 5 loops over all undeleted clauses in the formula. Formulas are represented as an array of clauses[6] together with a lazily updated list that tracks all indices of the array containing undeleted clauses. This enables both constant-time lookup of clauses throughout the algorithm and fast iteration over the undeleted clauses for Step 5. Deletion in the index list is done in (amortized) constant time by removing a deleted index only when the index is looked up in Step 5.1.

Additionally, for each literal, the smallest clause index where that literal occurs (if any) is lazily tracked in a lookup array; for a given witness ω, all clauses occurring at indices below the index of any literal in $\overline{\omega}$ can be skipped in Step 5.1.

5.3.2 Proof checking exceptions

There are several steps in the proof checking algorithm that can fail (report `NO`) if the input proof is invalid, e.g., in Step 3.3. In a purely functional implementation, results are represented with an option: None indicating a failure and Some *res* indicating success with result *res*. While conceptually simple, this means that common case (successful) intermediate results are always boxed within a Some constructor and then immediately unboxed with pattern matching to be used again. In `cake_lpr`, failures instead raise exceptions which are directly handled at the top level. Thus, successful results can be passed directly, i.e., as *res*, without any boxing. Support for verifying the use of exceptions is a unique feature of CakeML's CF framework [19].

5.3.3 Hashtables

For compositional proof checking (Fig. 5), it is important to check the inclusion of clauses $F_j \subseteq G$ efficiently since both formulas can contain a large number of clauses. To achieve this, the formula G (in the array-based representation) is converted into CakeML's verified hashtable library using a simple rolling hash for clauses. This allows every clause in F_j to be checked against the hashtable for G in near-constant time.

5.3.4 Buffered I/O streams

Proof files generated by SAT solvers can be large, e.g., ranging from 300 MB to 4 GB for the second benchmark suite in Section 6. These files are streamed into memory line by line because each proof step depends only on information contained in its corresponding line in the file. This streaming interaction is optimized using CakeML's verified buffered I/O library [41] which maintains an internal buffer of yet-to-be-read bytes from the read-only proof file to batch and minimize the number of expensive filesystem I/O calls.

5.3.5 Producing MD5 hashes

As part of this work, we verified a CakeML library for computing the MD5 hash of an input stream which is connected with the buffered I/O library to efficiently compute hashes for the compositional proof checking command-line options 4 and 5 in Section 5.1. The verification shows that our imperative implementation of MD5 hashing matches its functional specification. In particular, since the MD5 hash is provided solely for user convenience, we do not prove any formal cryptographic properties of the hashing function.

6 Benchmarks

This section compares the CakeML LPR proof checker against other verified checkers on two benchmark suites (Sections 6.1 and 6.2) and a RAT microbenchmark (Section 6.3). It also evaluates the compositional proof format on unsatisfiability proofs of Erdős discrepancy properties [35] (Section 6.4). The first suite is a collection of problems with PR proofs generated by the *satisfaction-driven clause learning* (SDCL) solver `SaDiCaL` [27]; the second suite consists of unsatisfiable problems from the SAT Race 2019 competition.[7] The RAT microbenchmark consists of proofs for large mutilated chessboards generated by a BDD-based SAT solver [8]. The unsatisfiability proofs for the Erdős discrepancy properties are generated by cube-and-conquer (see Section 4.2). Raw timing data used to produce the figures and tables in this section is available from the `cake_lpr` repository.

[6] Deleted clauses are no longer referenced by the array and are automatically freed by CakeML's garbage collector.

[7] The suites are available at http://fmv.jku.at/sadical/ and http://sat-race-2019.ciirc.cvut.cz/ respectively. Note that `cake_lpr` was also used to validate proofs for the 2020 SAT Competition (results not reported here).

The CakeML checker is labeled `cake_lpr` (default 4GB heap and stack space), while the other proof checkers used are labeled `acl2-lrat` (verified in ACL2 [20]), `coq-lrat` (verified in Coq [10]), and `GRATchk` (verified in Isabelle/HOL [39]) respectively. All experiments were ran on identical nodes with Intel Xeon E5-2695 v3 CPUs (35M cache, 2.30GHz) and 128GB RAM; more specific configuration options for each benchmark suite are reported below.

6.1 SaDiCaL PR Benchmarks

The SaDiCaL solver produces PR proofs for hard SAT problems in its benchmark suite [27] and it is experimentally much faster than a plain DRAT-based CDCL solver on those problems [27, Section 7]. The PR proofs are directly checked by `cake_lpr` after conversion into LPR format with `DPR-trim`. For all other checkers, the PR proofs were first converted to DRAT format using `pr2drat` (as in an earlier approach [27]), and then into LRAT and GRAT formats using the `DRAT-trim` and `GRATgen`[8] tools respectively. All tools were ran with a timeout of 10000 seconds and all timings are reported in seconds (to one d.p.). Results are summarized in Tables 2 and 3.

All benchmarks were successfully solved by SaDiCaL except `mchess19` which exceeded the time limit. For the remaining benchmarks, generating and checking LPR proofs required a comparable (1–2.5x) amount of time to solving the problems, except `mchess`, for which LPR generation and checking is much faster than solving the problems (Table 2). Unsurprisingly, direct checking of LPR proofs is *much faster* than the circuitous route of converting into DRAT and then into either LRAT or GRAT (Table 3). Unlike LPR, checking PR proofs via the LRAT route is 5–60x slower than solving those problems; this is a significant drawback to using the route in practice for certifying solver results. Compared to an unverified C implementation of LPR proof checking, `cake_lpr` is approximately an order of magnitude slower on these benchmarks; a detailed comparison against unverified proof checking is in Section 6.3.

The backwards compatibility of `cake_lpr` is also shown in Table 3, where it is used to check the generated LRAT proofs. Among the LRAT checkers, `acl2-lrat` is fastest, followed by `cake_lpr` (LRAT checking), and `coq-lrat`. Although `cake_lpr` (LRAT checking) is on average 1.3x slower than `acl2-lrat`, it scales better on the `mchess` problems and is actually much faster than `acl2-lrat` on `mchess18`. We also observed that the GRAT toolchain (summing `SaDiCaL`, `pr2drat`, `GRATgen` and `GRATchk` times) is much slower than the LRAT toolchains (summing `SaDiCaL`, `pr2drat`, `DRAT-trim` and fastest LRAT checking times). This is in contrast to the SAT Race 2019 benchmarks below (see Fig. 12), where we observed the opposite relationship. We believe that the difference in checking speed is due to the various checkers having different optimizations for checking the expensive RAT proof steps produced by conversion from PR proofs.

6.2 SAT Race 2019 Benchmarks

We further benchmarked the verified checkers on a suite of 102 unsatisfiable problems from the SAT Race 2019 competition.[9] For all problems, DRAT proofs were generated using the state-of-the-art SAT solver `CaDiCaL` before conversion into the LRAT or GRAT formats. Proofs generated by `CaDiCaL` on this suite rarely require RAT (or PR) steps, so the checkers are stress-tested on their implementation of file I/O, parsing, and reverse unit propagation based on the annotated hints (Step 3.1 from Fig. 3); `cake_lpr` is the *only* tool with a formally verified implementation of the former two steps. All tools were ran with the SAT competition solver timeout of 5000 seconds.

A summary of the results is given in Table 4, where all proofs generated by `CaDiCaL` were checked by at least one verified checker. The `acl2-lrat` checker fails with a parse error on one problem even though none of the other checkers reported such an error; `GRATgen` aborted on two problems for an unknown reason. Plots comparing LRAT proof checking time and overall proof generation and checking time (LRAT and GRAT) are shown in Fig. 12. From Fig. 12 (top), the relative order of LRAT checking speeds remains the same as Table 3, where `cake_lpr` is on average 1.2x slower than `acl2-lrat`, although `cake_lpr` is faster on 28 benchmarks. From Fig. 12 (bottom), both LRAT toolchains are slower than the GRAT toolchain (average 3.5x slower for `cake_lpr` and 3.4x for `acl2-lrat`). Part of this speedup for the GRAT toolchain comes from `GRATgen` which can be run in parallel (with 8 threads). This suggests that adding native support for GRAT-based input to `cake_lpr` or adding LRAT support to `GRATgen` could be worthwhile future extensions.

[8] The `GRATgen` tool supports parallel proof generation and was ran with 8 threads.

[9] There are 117 problems from the competition which are known to be unsatisfiable; `CaDiCaL` proved unsatisfiability for 102 problems within the 5000 seconds timeout.

Table 2 Timings for PR benchmarks with conversion into LPR format. The "Total (LPR)" column sums the generation and checking times. The timing for `mchess19` is omitted because `SaDiCaL` timed out; timings for the Urquhart `U.-s3-*` benchmarks are omitted because they took a negligible amount of time (< 1.0s total).

Problem	SaDiCaL	DPR-trim	cake_lpr (LPR)	Total (LPR)	Problem	SaDiCaL	DPR-trim	cake_lpr (LPR)	Total (LPR)
hole20	1.0	0.5	0.7	2.2	U.-s4-b1	0.7	0.6	0.3	1.6
hole30	6.9	2.4	6.1	15.4	U.-s4-b2	0.3	0.4	0.2	0.8
hole40	31.3	10.0	25.1	66.3	U.-s4-b3	0.4	0.4	0.2	1.0
hole50	101.7	35.5	87.9	225.1	U.-s4-b4	0.3	0.5	0.3	1.1
mchess15	18.5	1.1	2.1	21.7	U.-s5-b1	2.5	0.9	1.3	4.7
mchess16	21.7	1.2	2.1	25.0	U.-s5-b2	1.2	0.6	0.7	2.4
mchess17	34.8	1.6	3.4	39.8	U.-s5-b3	3.2	1.5	2.0	6.8
mchess18	59.8	2.3	5.2	67.2	U.-s5-b4	5.5	1.5	3.2	10.1

Table 3 Timings for PR benchmarks, first converted to DRAT and subsequently converted into LRAT and GRAT formats. The "Total (LRAT)" and "Total (GRAT)" columns sum the fastest generation and checking times for the LRAT and GRAT formats respectively. The "Total (LPR)" column (in **bold**, fastest total time) is reproduced from Table 2 for ease of comparison. Fail(T) indicates a timeout. Timings for the `mchess19` and `U.-s3-*` benchmarks are omitted as in Table 2.

Prob.	pr2drat	DRAT-trim	cake_lpr (LRAT)	acl2-lrat	coq-lrat	GRATgen	GRATchk	**Total (LPR)**	Total (LRAT)	Total (GRAT)
hole20	0.8	4.4	18.5	7.9	966.7	4.6	18.2	**2.2**	14.2	24.6
hole30	6.8	61.4	180.4	105.9	Fail(T)	24.5	647.9	**15.4**	181.0	686.1
hole40	32.4	460.0	1039.5	711.8	Fail(T)	101.3	Fail(T)	**66.3**	1235.5	-
hole50	108.6	2663.0	4697.4	3292.2	Fail(T)	337.2	Fail(T)	**225.1**	6165.5	-
mchess15	7.7	48.2	49.3	36.2	Fail(T)	48.4	2023.1	**21.7**	110.6	2097.7
mchess16	9.0	62.0	59.8	53.2	Fail(T)	55.2	2903.8	**25.0**	145.9	2989.6
mchess17	14.5	105.0	97.3	88.5	Fail(T)	86.1	7050.9	**39.8**	242.7	7186.3
mchess18	25.1	195.0	152.7	296.8	Fail(T)	135.9	Fail(T)	**67.2**	432.5	-
U.-s4-b1	0.5	2.5	3.6	3.3	135.7	3.6	44.8	**1.6**	7.0	49.7
U.-s4-b2	0.2	0.8	1.4	1.0	23.2	1.7	8.2	**0.8**	2.3	10.4
U.-s4-b3	0.3	1.3	2.0	1.5	49.2	2.4	16.2	**1.0**	3.5	19.3
U.-s4-b4	0.3	1.1	1.8	1.4	38.3	2.0	10.3	**1.1**	3.1	12.9
U.-s5-b1	4.2	13.6	16.7	12.5	3048.7	17.4	933.2	**4.7**	32.8	957.3
U.-s5-b2	1.7	5.6	7.3	5.5	614.7	7.7	189.6	**2.4**	13.9	200.2
U.-s5-b3	5.0	18.4	26.3	22.2	8750.5	21.1	2316.3	**6.8**	48.8	2345.6
U.-s5-b4	11.3	34.2	36.9	30.1	Fail(T)	40.6	Fail(T)	**10.1**	81.0	-

Table 4 A summary of the SAT Race 2019 benchmark results on 102 unsatisfiability proofs generated by `CaDiCaL`. The N/A row counts problems that timed out or failed in an earlier step of the respective toolchains.

Status	DRAT-trim	acl2-lrat	cake_lpr	coq-lrat	GRATgen	GRATchk
Success	97	96	97	36	100	100
Timeout	5	0	0	61	0	0
Other Failures	0	1	0	0	2	0
N/A	0	5	5	5	0	2

6.3 Mutilated Chessboard RAT Microbenchmarks

The following microbenchmark suite tests the LRAT checkers on large mutilated chessboard problems (up to 100 by 100) solved by `pgbdd`, a BDD-based SAT solver [8]. Unlike the previous benchmark suites, LRAT proofs are emitted *directly* by the solver so additional `DRAT-trim` conversion is not needed. All tools were ran with a timeout of 10000 seconds and all timings are reported in seconds (to one d.p.). For additional scaling comparison, we also report results for `lrat-check`, an *unverified* LRAT proof checker implemented in C.

The results in Table 5 show the impact of `cake_lpr`'s RAT optimizations (Section 5.3). Notably, `cake_lpr` scales essentially *linearly* in the size of the proofs (up to ≈ 10 million proof steps). As a result, `cake_lpr` is significantly faster than `acl2-lrat` and `coq-lrat` on these RAT-heavy proofs and it comes within a 5x factor of the unverified `lrat-check` tool.

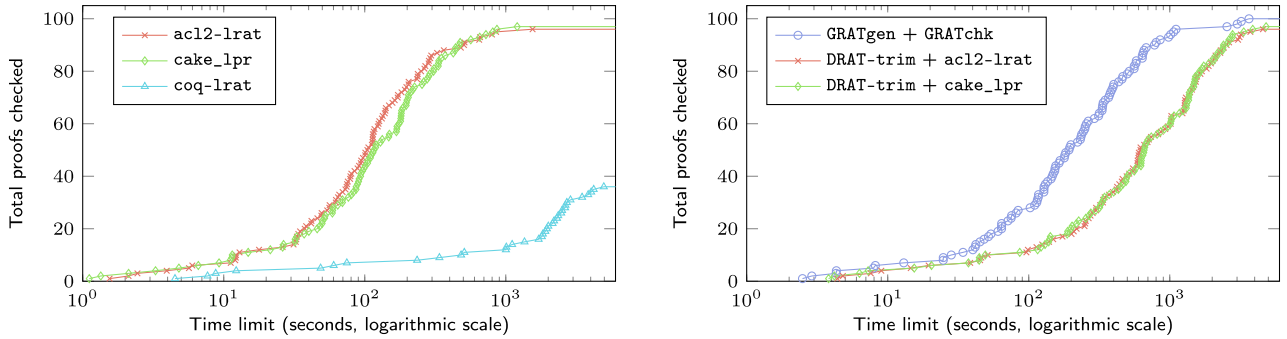

Fig. 12 (Left) SAT Race 2019 proofs checked within a given (per instance) time limit for the LRAT proof checkers. (Right) SAT Race 2019 proofs generated and checked within a given (per instance) time limit for the LRAT and GRAT toolchains.

Table 5 Timings for the RAT microbenchmark. The number of proof steps and file size of the proofs (in MB) are shown in the last two columns. Fail(T) indicates a timeout.

Problem	pgbdd	lrat-check	cake_lpr	acl2-lrat	coq-lrat	LRAT Steps	File Size
mchess20	3.9	0.5	0.5	19.6	3405.2	125752	5
mchess40	47.5	1.0	3.5	453.4	Fail(T)	769287	36
mchess60	311.7	2.7	10.6	4885.2	Fail(T)	2300522	114
mchess80	1164.1	4.8	22.6	Fail(T)	Fail(T)	5089457	259
mchess100	3599.0	9.3	44.2	Fail(T)	Fail(T)	9506092	499

6.4 Erdős Discrepancy Properties

The Erdős Discrepancy Problem (EDP) asks if, for every $C > 0$ and infinite ± 1 sequence $(x_1, x_2, x_3, \ldots)$, there exists k, d such that $|\sum_{i=1}^{k} x_{i \cdot d}| > C$. Konev and Lisitsa [35] provided a SAT-solver aided proof of the EDP in the case $C = 2$. We demonstrate the parallel scalability of compositional proof checking (Section 4) using `cake_lpr`'s compositional proof checking option on a cube-and-conquer proof generated by `iGlucose`, a version of `Glucose 3.0` with support for iCNF files, and the technique reported in Section 4.2. The technique yields a 5226 line compositional top-level proof where each line is justified by an underlying LRAT proof, see Fig. 8. The underlying proofs consist of 20 million clause addition lines in total and they vary widely in size, ranging from 88 bytes to 110 MB.

Proof steps are allocated evenly to parallel threads by their index. For example, with two threads, the first thread would check all odd-numbered proof steps by sequentially running instances of `cake_lpr` to check the underlying proofs for ranges $(i_0, i_1), (i_2, i_3), \ldots$, while the second thread checks all even-numbered proof steps for ranges $(i_1, i_2), (i_3, i_4), \ldots$, and similarly for more parallel threads. Results are summarized in Table 6 with wall-clock execution times reported in seconds and relative speedup against a single thread. All values are rounded to one decimal place.

The speedup is nearly linear for lower number of threads (1–32) but drops off at 64 and 128 threads. This is likely due to the unbalanced nature of proof sizes, where large LRAT proofs dominate the overall proof checking time at high levels of parallelism. A more advanced parallelization scheme, e.g., with a scheduler, could further improve proof checking performance.

Table 6 Timings (wall-clock) for the EDP benchmark. Speedup is calculated relative to 1 thread.

Threads	cake_lpr	Speedup
1	3871.7	-
2	1985.4	2.0
4	998.2	3.9
8	517.4	7.5
16	273.2	14.2
32	136.4	28.4
64	76.5	50.6
128	60.2	64.3

7 Related Work

Verified Proof Checking. There are several RAT-based verified proof checkers, in ACL2 [20], Coq [10], and Isabelle/HOL [39]; these verified checkers all use an unverified preprocessing tool to add proof hints to DRAT proofs, either `DRAT-trim` or `GRATgen`. Alternative preprocessing tools are available, e.g., based on the recently proposed FRAT format [4]. The DRAT format is itself an extension of the

DRUP format [21]; the Coq checker is based on a predecessor verified checker for the GRIT [11] format. The ACL2 checker can be efficiently and *directly executed* (without extraction) using imperative primitives native to the ACL2 kernel [20]. However, the implementation of these features in ACL2 itself must be trusted to trust the proof checking results [50], hence the yellow background in Table 1. SMT-Coq [2, 14] is another certificate-based checker for SAT and SMT problems in Coq. Its resolution-based proof certificates can be checked natively using native computation extensions of the Coq kernel.

Verified checkers are available for other logics, such as the Verified TESC Verifier for first-order logic [3], Pastèque for the practical algebraic calculus [33], and various checkers for higher-order logics (and type theories) underlying proof assistants [1, 47, 52].

Applications. SAT solving is a key technology underlying many software and hardware verification domains [7, 30]. Certifying SAT results adds a layer of trust and is clearly a worthwhile endeavor. Solver-aided proofs of mathematical results [23, 29, 35] are particularly interesting and challenging to certify because these often feature complicated SAT encodings, custom (hand-crafted) proof steps, and enormous resulting proofs [29]. Our `cake_lpr` checker is designed to handle the latter two challenges effectively. For the first challenge, the SAT encoding of mathematical problems can also be verified within proof assistants. This was done for the Boolean Pythagorean Triples problem building on the Coq proof checker [12].

Verified SAT Solving. An alternative to proof checking is to verify the SAT solvers [16, 17, 43, 48]. This is a significant undertaking but it would allow the pipeline of generating and checking proofs to be entirely bypassed. Furthermore, such verification efforts can yield new insights about key invariants underlying SAT solving techniques compared to prior pen-and-paper presentations, e.g., the 2WL invariant [17]. However, the performance of verified SAT solvers are not yet competitive with modern (unverified) SAT solving technology [16, 17].

8 Conclusion

This work presents the LPR proof format for verified checking of PR proofs and a compositional proof format for separate (parallel) proof checking. It also demonstrates the feasibility of using binary code extraction to verify `cake_lpr`, a performant proof checker supporting both formats down to its machine-code implementation. Given the strength of the PR proof system, there is ongoing research into the design of *satisfaction-driven clause learning* techniques [27, 28, 49] for SAT solvers based on PR clauses. Our proof checker opens up the possibility of using a verified checker to help check and debug the implementation of these new techniques. It also gives future SAT competitions the option of providing PR as the default (verified) proof system for participating solvers. Another interesting direction is to add `cake_lpr` support for other proof formats [5, 39]. In particular, this would allow users to build compositional proofs that utilize different underlying proof formats for separate parts of the proof.

Acknowledgements We thank the anonymous reviewers for their helpful feedback on this article, as well as Jasmin Blanchette and the TACAS'21 anonymous reviewers for their helpful feedback on the earlier conference version. We also thank Peter Lammich for help with `GRATgen` and Stefan O'Rear for help with profiling CakeML programs.

Funding Open access funding provided by Chalmers University of Technology.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abrahamsson, O.: A verified proof checker for higher-order logic. J. Log. Algebraic Methods Program. **112**, 100530 (2020). https://doi.org/10.1016/j.jlamp.2020.100530
2. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of SAT/SMT solvers to Coq through proof witnesses. In: Jouannaud, J., Shao, Z. (eds.) CPP, LNCS, vol. 7086, pp. 135–150. Springer (2011). https://doi.org/10.1007/978-3-642-25379-9_12
3. Baek, S.: A formally verified checker for first-order proofs. In: Cohen, L., Kaliszyk, C. (eds.) ITP, LIPIcs, vol. 193, pp. 6:1–6:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.6
4. Baek, S., Carneiro, M., Heule, M.J.H.: A flexible proof format for SAT solver-elaborator communication. Log. Methods Comput. Sci. (2022). https://doi.org/10.46298/lmcs-18(2:3)2022
5. Barnett, L.A., Biere, A.: Non-clausal redundancy properties. In: Platzer, A., Sutcliffe, G. (eds.) CADE, LNCS, vol. 12699, pp. 252–272. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_15
6. Becker, H., Zyuzin, N., Monat, R., Darulova, E., Myreen, M.O., Fox, A.C.J.: A verified certificate checker for finite-precision

error bounds in Coq and HOL4. In: Bjørner, N., Gurfinkel, A. (eds.) FMCAD, pp. 1–10. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603019
7. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, R. (ed.) TACAS, LNCS, vol. 1579, pp. 193–207. Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
8. Bryant, R.E., Heule, M.J.H.: Generating extended resolution proofs with a BDD-based SAT solver. In: Groote, J.F., Larsen, K.G. (eds.) TACAS, LNCS, vol. 12651, pp. 76–93. Springer (2021). https://doi.org/10.1007/978-3-030-72016-2_5
9. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: VST-Floyd: a separation logic tool to verify correctness of C programs. J. Autom. Reason. **61**(1–4), 367–422 (2018). https://doi.org/10.1007/s10817-018-9457-5
10. Cruz-Filipe, L., Heule, M.J.H., Hunt Jr., W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) CADE, LNCS, vol. 10395, pp. 220–236. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_14
11. Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: Legay, A., Margaria, T. (eds.) TACAS, LNCS, vol. 10205, pp. 118–135 (2017). https://doi.org/10.1007/978-3-662-54577-5_7
12. Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Formally verifying the solution to the Boolean Pythagorean triples problem. J. Autom. Reason. **63**(3), 695–722 (2019). https://doi.org/10.1007/s10817-018-9490-4
13. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT, LNCS, vol. 2919, pp. 502–518. Springer (2003). https://doi.org/10.1007/978-3-540-24605-3_37
14. Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A., Barrett, C.W.: SMTCoq: a plug-in for integrating SMT solvers into Coq. In: Majumdar, R., Kuncak, V. (eds.) CAV, LNCS, vol. 10427, pp. 126–133. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_7
15. Férée, H., Pohjola, J.Å., Kumar, R., Owens, S., Myreen, M.O., Ho, S.: Program verification in the presence of I/O - semantics, verified library routines, and verified applications. In: Piskac, R., Rümmer, P. (eds.) VSTTE, LNCS, vol. 11294, pp. 88–111. Springer (2018). https://doi.org/10.1007/978-3-030-03592-1_6
16. Fleury, M.: Optimizing a verified SAT solver. In: Badger, J.M., Rozier, K.Y. (eds.) NFM, LNCS, vol. 11460, pp. 148–165. Springer (2019). https://doi.org/10.1007/978-3-030-20652-9_10
17. Fleury, M., Blanchette, J.C., Lammich, P.: A verified SAT solver with watched literals using Imperative HOL. In: Andronick, J., Felty, A.P. (eds.) CPP, pp. 158–171. ACM (2018). https://doi.org/10.1145/3167080
18. Ghale, M.K., Pattinson, D., Kumar, R., Norrish, M.: Verified certificate checking for counting votes. In: Piskac, R., Rümmer, P. (eds.) VSTTE, LNCS, vol. 11294, pp. 69–87. Springer (2018). https://doi.org/10.1007/978-3-030-03592-1_5
19. Guéneau, A., Myreen, M.O., Kumar, R., Norrish, M.: Verified characteristic formulae for CakeML. In: Yang, H. (ed.) ESOP, LNCS, vol. 10201, pp. 584–610. Springer (2017). https://doi.org/10.1007/978-3-662-54434-1_22
20. Heule, M., Hunt Jr., W.A., Kaufmann, M., Wetzler, N.: Efficient, verified checking of propositional proofs. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP, LNCS, vol. 10499, pp. 269–284. Springer (2017). https://doi.org/10.1007/978-3-319-66107-0_18
21. Heule, M., Hunt Jr., W.A., Wetzler, N.: Trimming while checking clausal proofs. In: FMCAD, pp. 181–188. IEEE (2013). https://doi.org/10.1109/FMCAD.2013.6679408
22. Heule, M., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC, LNCS, vol. 7261, pp. 50–65. Springer (2011). https://doi.org/10.1007/978-3-642-34188-5_8
23. Heule, M.J.H.: Schur number five. In: McIlraith, S.A., Weinberger, K.Q. (eds.) AAAI, pp. 6598–6606. AAAI Press (2018)
24. Heule, M.J.H., Biere, A.: Compositional propositional proofs. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR, LNCS, vol. 9450, pp. 444–459. Springer (2015). https://doi.org/10.1007/978-3-662-48899-7_31
25. Heule, M.J.H., Biere, A.: What a difference a variable makes. In: Beyer, D., Huisman, M. (eds.) TACAS, LNCS, vol. 10806, pp. 75–92. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_5
26. Heule, M.J.H., Kiesl, B., Biere, A.: Clausal proofs of mutilated chessboards. In: Badger, J.M., Rozier, K.Y. (eds.) NFM, LNCS, vol. 11460, pp. 204–210. Springer (2019). https://doi.org/10.1007/978-3-030-20652-9_13
27. Heule, M.J.H., Kiesl, B., Biere, A.: Encoding redundancy for satisfaction-driven clause learning. In: Vojnar, T., Zhang, L. (eds.) TACAS, LNCS, vol. 11427, pp. 41–58. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_3
28. Heule, M.J.H., Kiesl, B., Biere, A.: Strong extension-free proof systems. J. Autom. Reason. **64**(3), 533–554 (2020). https://doi.org/10.1007/s10817-019-09516-0
29. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the boolean Pythagorean triples problem via cube-and-conquer. In: Creignou, N., Berre, D.L., (eds.) SAT, LNCS, vol. 9710, pp. 228–245. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_15
30. Jackson, D., Schechter, I., Shlyakhter, I.: Alcoa: the Alloy constraint analyzer. In: Ghezzi, C., Jazayeri, M., Wolf, A.L. (eds.) ICSE, pp. 730–733. ACM (2000). https://doi.org/10.1145/337180.337616
31. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR, LNCS, vol. 7364, pp. 355–370. Springer (2012). https://doi.org/10.1007/978-3-642-31365-3_28
32. Kanabar, H.: Implementing and verifying a compiler optimisation for CakeML (2018). https://hrutvik.co.uk/assets/pdf/Hrutvik_Kanabar_dissertation.pdf. Computer Science Tripos, Part II Dissertation. University of Cambridge, UK
33. Kaufmann, D., Fleury, M., Biere, A.: The proof checkers Pacheck and Pastèque for the practical algebraic calculus. In: FMCAD, pp. 264–269. IEEE (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_34
34. Kiesl, B., Rebola-Pardo, A., Heule, M.J.H.: Extended resolution simulates DRAT. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR, LNCS, vol. 10900, pp. 516–531. Springer (2018). https://doi.org/10.1007/978-3-319-94205-6_34
35. Konev, B., Lisitsa, A.: Computer-aided proof of Erdős discrepancy properties. Artif. Intell. **224**, 103–118 (2015). https://doi.org/10.1016/j.artint.2015.03.004
36. Kumar, R., Mullen, E., Tatlock, Z., Myreen, M.O.: Software verification with ITPs should use binary code extraction to reduce the TCB - (short paper). In: Avigad, J., Mahboubi, A. (eds.) ITP, LNCS, vol. 10895, pp. 362–369. Springer (2018). https://doi.org/10.1007/978-3-319-94821-8_21
37. Lammich, P.: Generating verified LLVM from Isabelle/HOL. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) ITP, LIPIcs, vol. 141, pp. 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.22
38. Lammich, P.: Refinement to Imperative HOL. J. Autom. Reason. **62**(4), 481–503 (2019). https://doi.org/10.1007/s10817-017-9437-1
39. Lammich, P.: Efficient verified (UN)SAT certificate checking. J. Autom. Reason. **64**(3), 513–532 (2020). https://doi.org/10.1007/s10817-019-09525-z
40. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814

41. Lind, J., Mihajlovic, N., Myreen, M.O.: Verified hash map and buffered I/O libraries for CakeML. In: Trends in Functional Programming (TFP) (2021). Accepted for presentation
42. Lööw, A., Kumar, R., Tan, Y.K., Myreen, M.O., Norrish, M., Abrahamsson, O., Fox, A.C.J.: Verified compilation on a verified processor. In: McKinley, K.S., Fisher, K. (eds.) PLDI, pp. 1041–1053. ACM (2019). https://doi.org/10.1145/3314221.3314622
43. Maric, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. **411**(50), 4333–4356 (2010). https://doi.org/10.1016/j.tcs.2010.09.014
44. Mullen, E., Pernsteiner, S., Wilcox, J.R., Tatlock, Z., Grossman, D.: Œuf: minimizing the Coq extraction TCB. In: Andronick, J., Felty, A.P. (eds.) CPP, pp. 172–185. ACM (2018). https://doi.org/10.1145/3167089
45. Myreen, M.O.: The CakeML project's quest for ever stronger correctness theorems (invited paper). In: Cohen, L., Kaliszyk, C. (eds.) ITP, LIPIcs, vol. 193, pp. 1:1–1:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.1
46. Myreen, M.O., Owens, S.: Proof-producing translation of higher-order logic into pure and stateful ML. J. Funct. Program. **24**(2–3), 284–315 (2014). https://doi.org/10.1017/S0956796813000282
47. Nipkow, T., Roßkopf, S.: Isabelle's metalogic: Formalization and proof checker. In: Platzer, A., Sutcliffe, G. (eds.) CADE, LNCS, vol. 12699, pp. 93–110. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_6
48. Oe, D., Stump, A., Oliver, C., Clancy, K.: versat: A verified modern SAT solver. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI, LNCS, vol. 7148, pp. 363–378. Springer (2012). https://doi.org/10.1007/978-3-642-27940-9_24
49. Reeves, J.E., Heule, M.J.H., Bryant, R.E.: Preprocessing of propagation redundant clauses. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR, LNCS, vol. 13385, pp. 106–124. Springer (2022). https://doi.org/10.1007/978-3-031-10769-6_8
50. Slind, K.: Trusted extensions of interactive theorem provers: Workshop summary (2010). https://www.cs.utexas.edu/users/kaufmann/itp-trusted-extensions-aug-2010/summary/summary.pdf. [Online; accessed 7-September-2021]
51. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs, LNCS, vol. 5170, pp. 28–32. Springer (2008). https://doi.org/10.1007/978-3-540-71067-7_6
52. Sozeau, M., Boulier, S., Forster, Y., Tabareau, N., Winterhalter, T.: Coq Coq correct! Verification of type checking and erasure for Coq. Coq. Proc. ACM Program. Lang. **4**(POPL), 8:1-8:28 (2020). https://doi.org/10.1145/3371076
53. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: cake_lpr: Verified propagation redundancy checking in CakeML. In: Groote, J.F., Larsen, K.G. (eds.) TACAS, LNCS, vol. 12652, pp. 223–241. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_12
54. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A.C.J., Owens, S., Norrish, M.: The verified CakeML compiler backend. J. Funct. Program. **29**, e2 (2019). https://doi.org/10.1017/S0956796818000229
55. Wetzler, N., Heule, M., Hunt Jr., W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT, LNCS, vol. 8561, pp. 422–429. Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_31
56. Wieringa, S., Niemenmaa, M., Heljanko, K.: Tarmo: A framework for parallelized bounded model checking. In: Brim, L., van de Pol, J. (eds.) PDMC, EPTCS, vol. 14, pp. 62–76 (2009). https://doi.org/10.4204/EPTCS.14.5

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.